A Metric Evaluation of Game Application Software

Alan C. Jamieson, Nicholas A. Kraft, Jason O. Hallstrom and Brian A. Malloy

Computer Science Department

Clemson University

Clemson, SC 29634, USA

{*ajamies,nkraft,jasonoh,malloy*}*@cs.clemson.edu*

Abstract

In this paper we evaluate the exploitation of object technology as it is used in a test suite of game application software. We use several well-known metrics and apply them to both the game application software and a test suite of language processing tools to form a basis of comparing the two groups. We have developed a metric computation system that uses the g^4re infrastructure to analyze the C++ programs. We present some results for the two groups of applications and draw some conclusions about the modularity, use of inheritance, and the complexity of methods in these applications.

1. Introduction

As software developers shift their priorities to the construction of complex large scale systems that are easy to extend, modify, and maintain, the inadequacy of traditional approaches and methodologies becomes apparent. The object-oriented approach to software development addresses some of these inadequacies and makes it possible to model systems that are close to their real world analogues. The goal of object-oriented design is to accurately identify the principle roles in a process, assign responsibilities to these roles and encapsulate them in an object. The benefits of object technology are extensibility, ease of modification and ease of reuse.

The development of game application software has followed this pattern of migration from traditional approaches to the exploitation of object technology for its acknowledged benefits. In fact, the object-oriented paradigm seems to have a natural application to the domain of graphics, Graphical User Interfaces (GUIs) and game application programming. Early games such as Quake and Doom were implemented in C because of its small learning curve and its fast compilation and execution speed. However, current game development requires large teams of programmers and analysts as well as other kinds of talent such as art, music, and voice. To manage large teams of programmers, most current game developers use the C++ language and attempt to exploit the benefits of object technology.

In this paper we evaluate the exploitation of object technology as it is used in a test suite of game application software. We use several well-known metrics and apply them to both the game application software and a test suite of language processing tools to form a basis of comparing the two groups. We have developed a metric computation system that uses the g^4re infrastructure to analyze the C++ programs [10, 11]. We present some results for the two groups of programs and draw some conclusions about the modularity, use of inheritance and the complexity of methods in the applications.

The literature on object-oriented software metrics is extensive; see, for example, the detailed surveys presented in [3] and [5]. By contrast, there has been relatively little work focused on applying metrics to assess the relative design characteristics of systems in different application domains. A unique contribution of our work is the application of object-oriented software metrics to the consideration of gaming applications.

In the next section we provide background about the terminology that we use, about game APIs, and about the g^4re system. In Section 3 we describe the construction of our metric computation system and define the metrics that we use to evaluate the exploitation of object orientation in game application software. In Section 4 we provide results from our evaluation of the test suite of game software and language processing tools. In Section 5 we describe the limited research that is similar to our work. Finally, in Section 6 we draw conclusions and describe our future work.

2. Background

In this section we provide background about the terminology and tools that we use in this paper. In Section 2.1 we review some of the Application Programmer Interfaces (APIs) used in game development. In Section 2.2 we review the g^4re system that analyzes the applications in our

test suite [10, 11]. Finally, in Section 2.3 we review the use of metrics in the software life cycle.

2.1. Game APIs

In early game development, DOS-based games were generally implemented with commands issued directly to the computer's hardware. These early DOS games used calls to device drivers for input devices such as a mouse or joystick and calls to specific sound cards such as Creative Labs' Sound Blaster. Video programming was the most difficult aspect of game development where the generation of fast and smooth graphics required significant programming skill. Graphics code frequently exploited the speed of assembly language programming and depended on certain hardware-level features of the VGA graphics adapter.

Currently, few game developers write register-level video code, instead relying on prewritten Application Programmer Interfaces (APIs) that form a layer of software between the game and the hardware. The most popular API in current usage is DirectX using the C++ language vehicle [15, 16]. The DirectX API provides low-level access to multimedia hardware in a device-independent manner. New versions of DirectX are released to permit game developers to take advantage of hardware advances as they occur, even after games have shipped. However, the DirectX API is specific to the Microsoft Windows platform.

With the popularity of the Linux operating system game developers became interested in platform-independent game programming and several APIs have been introduced, including SVGALib, ClanLib and SDL. The most popular of the platform-independent game APIs is the Simple Directmedia Layer (SDL) [18]. SDL is a cross-platform multimedia library that has already been used to port a number of Windows-based games to Linux.

The SDL API supports virtually all of the major operating systems including Linux, Windows, Solaris and BSD variants including FreeBSD and MacOS. In addition to fast graphics support, SDL provides interfaces for playing sound, accessing CD-ROM drives and achieving portable multi-threaded applications. SDL is released under the GNU LGPL and has accumulated a collection of user-contributed libraries that provide additional functionality for game developers.

2.2. The g^{4}re Tool Chain

Software tools are fundamental to the comprehension, analysis, testing and debugging of application systems. Tools can automate repetitive tasks and, with large scale systems, can enable computation that would be prohibitively time consuming if performed manually. The Java language is well supported with libraries and tools to support application development [4, 23, 25]. The lack of tool support for applications using the C++ language is especially noteworthy.

One explanation for the lack of software tools for C++ is the difficulty in constructing a front-end for the language, as described in references [2, 9, 12, 20, 21, 22]. This difficulty results, in part, from the complexity and scale of the language. However, a more important problem is the ambiguity inherent in many C++ language constructs[9, 12, 19, 22]. Many C++ constructs cannot be recognized through syntactic considerations alone. For example, the difficulty in distinguishing a declaration from an expression can only be resolved by performing name lookup[1, 9].

The g^{4}re tool chain exploits the *gcc* Abstract Semantic Graph (ASG), GENERIC, to provide an Application Programmers Interface (API) to facilitate easy access to information about declarations, including classes, functions, and variables, as well as information about scopes, types, and control statements. The advantages of the g^{4}re tool chain is that it can analyze any program that can be compiled by the *gcc* C++ compiler. We use the g^{4}re tool chain to construct our metric computation system and we describe this system in Section 3.

2.3. Object-Oriented Metrics

Software metrics are quantitative measures that enable software developers, testers, and maintainers to evaluate the static properties of a software system [5]. Software metrics are computed and the resultant data are collected, analyzed, and compared throughout the lifetime of a software system to evaluate improvement or deterioration of the software system. Software metrics are also useful for identifying problem modules of a software system.

Object-oriented metrics were introduced to measure software properties specific to object-oriented software systems, including properties pertaining to classes and their object instances [3, 5]. The primary focus of object-oriented metrics is measuring properties of classes and their instances. Properties of interest include scope of properties, object complexity, coupling, and cohesion.

3. Methodology

In this section we present an overview of our metrics computation system that enables us to evaluate the exploitation of object technology in game application software. In Section 3.1 we describe details of the system and its use of the g^{4}re tool chain [10, 11]. In Section 3.2 we describe the metrics that we compute to facilitate our evaluation of the game software.
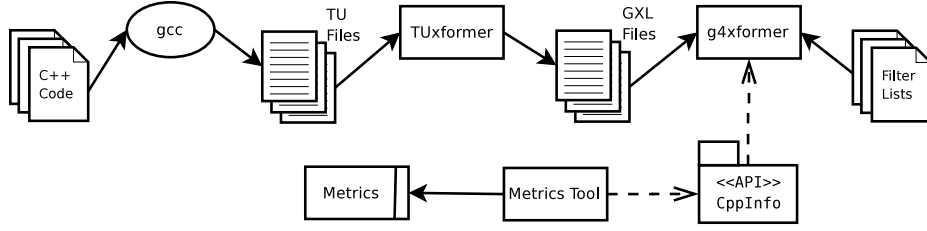
Figure 1. System overview. This figure illustrates the important components in our metrics computation system that we constructed to compute the metrics defined in Section 3.2 and to perform the case study presented in Section 4. The metrics computation system consists of the g^{4}re tool chain, including the CppInfo **API, and a** Metrics Tool **that interacts with the API to extract information about a C++ program.**

3.1. Overview of the Metrics Computation System

Figure 1 provides an overview of our metrics computation system that we constructed to compute the metrics defined in Section 3.2 and to perform the case study presented in Section 4. Our metrics computation system consists of the g^{4}re tool chain, including the CppInfo API, and a Metrics Tool that interacts with the API to extract information about a C++ program. Output of our system is a set of statistics for each computed metric.

Input to our system is the source code for a C++ program, shown in the far left of the top row of the figure, which is used as input to the *gcc* compiler. Using the -fdump-translation-unit-all option, we obtain a plain text representation of the ASG for each C++ translation unit in the program. We use these plain text ASG representations, known as tu files, as input to our TUxformer subsystem, shown in the middle of the top row of the figure. For each tu file, the TUxformer subsystem creates an in-memory representation of the encoded ASG, prunes the ASG, and serializes the ASG to GXL.

We use the set of GXL files produced by TUxformer as input to the g4xformer subsystem, shown in the far right of the top row of the figure. The g4xformer subsystem parses each GXL file and creates an in-memory representation of the encoded ASG. The subsystem then links the representations of each individual ASG to create a unified representation of the entire C++ program. After linking is complete, the subsystem filters, from the unified representation, language elements that are identified as defined in a filename contained in the filter lists, shown in the far right of the top row of the figure.

The CppInfo API, shown in the far right of the bottom row of the figure, provides access to information from the unified representation of a whole C++ program created by the g4xformer subsystem. Our Metrics Tool, shown in the middle of the bottom row of the figure, instantiates and queries the API to gain access to the information about

classes and functions needed to compute the metrics defined in Section 3.2. Output of the Metrics Tool, shown in the far left of the bottom row of the figure, is available in a variety of formats and consists of a set of statistics for each computed metric.

3.2. Overview of the Computed Metrics

In this section we define the object-oriented metrics that we compute for the case study presented in Section 4. We define one metric measuring complexity, Weighted Methods per Class, and three metrics measuring the use of inheritance: Depth of Inheritance Tree, Number of Ancestors, and Number of Children. We also use additional metrics that measure modularity and delegation.

3.2.1. Metric 1: Weighted Methods per Class (WMC). WMC measures the complexity of an object and is an indicator of the time and effort required to develop and maintain a class.

Given a class C with methods $M_1, M_2, \ldots, M_n$, weighted with cyclomatic complexity $c_1, c_2, \ldots, c_n$, respectively, the metric is computed as

$$WMC(C) = \sum_{i=1}^{n} c_i$$

Given a method M with control flow graph $G = (V, E)$, let D equal the set of decision nodes in V, where a decision node represents one of { *if, switch, for, while, do while, catch* }. The cyclomatic complexity, c, of M is the number of linearly independent paths in G and is computed as

$$c(M) = |D| + 1$$

3.2.2. Metric 2: Depth of Inheritance Tree (DIT). DIT is the length of the maximum path from a class to the root of its inheritance hierarchy, relates to scope of properties, and is an indicator of the number of ancestor classes that can potentially affect a class.

	SDL Game Applications				Language Processing Applications			
	ASC	AvP	Freespace2	Scorched3D	Doxygen	g^{4}re	Jikes	Keystone
Version	1.16.1.0	cvs	cvs	38.1	1.3.9.1	1.0.4	1.22	0.2.3
Source Files	436	509	652	1069	260	128	75	123
Translation Units	199	222	220	513	122	60	38	52
C++ Translation Units	194	95	220	492	90	60	38	52
LOC ($\approx$)	130 K	318 K	365 K	110 K	200 K	10 K	70 K	30 K

Table 1. *Testsuite of SDL Game Application Software and Language Processing Tools.*

Given a class *C* with a set of base classes *BC*, the metric is computed as

$$DIT(C) = \begin{cases} 0 & \text{if } |BC| = 0 \\ max(\{DIT(B_i) : B_i \in BC\}) + 1 & \text{if } |BC| > 0 \end{cases}$$

3.2.3. Metric 3: Number of Ancestors (NOA). NOA is the total number of ancestor classes of a class. In the absence of multiple inheritance, NOA is equivalent to DIT. In the presence of multiple inheritance, care must be taken to avoid counting an ancestor class more than once, due to the possibility of a diamond-shaped inheritance hierarchy.

3.2.4. Metric 4: Number of Children (NOC). NOC is the number of immediate successors of a class and measures the breadth of inheritance.

Given a class *C* with a set of derived classes *DC*, the metric is computed as

$$NOC(C) = |DC|$$

4. Case Study

In this section we describe the results that we obtained using our metrics tool to evaluate game application software. We evaluate game software by comparing metrics computed for four games with four popular language processing applications. The results that we report in this section capture information about the sizes of the programs and the exploitation of object technology as measured by metrics described in Section 3. All experiments were executed on a workstation with an *AMD Athlon64 3000+* processor, 1024 MB of PC3200 DDR RAM, and a 7200 RPM SATA hard drive, running the Slackware 10.1 operating system. The programs were compiled using *gcc* version 3.3.4.

In the next section we describe the eight (8) applications that form our test suite: four game applications implemented using the SDL API and four language processing applications, and provide some results using coarse-grained size metrics for the applications. In Section 4.2 we provide results describing the modularity of game applications and in Section 4.3 we provide results describing the

use of inheritance in game applications. In Section 4.4 we provide results describing the complexity of member functions in classes. Finally, in Section 4.5 we provide results for games implemented in DirectX.

4.1. The Test suite of Game Applications and Language Processing Tools

Table 1 lists eight applications, or test cases, that form the test suite that we use in our study, together with size statistics about each test case. The top row of the table lists the names that we use to refer to each of the test cases. The game applications are listed in the first four columns and the language processing applications are listed in the last four columns. The four game applications are: *Allied Strategic Command* (ASC), *Alien vs Predator* (AvP), *Freespace 2* (Freespace2), and *Scorched 3d* (Scorched3D). The Application Programmer's Interface (API) used for the four games is the Simple Directmedia Layer (SDL), described in Section 2. The four language processing applications, listed in the last four columns of Table 1, are: *Doxygen*, *g^{4}re*, *Jikes*, and *Keystone*. *Doxygen* is a documentation system for C++, C, and Java [26] and *g^{4}re* is part of the infrastructure for reverse engineering that we use to construct our metrics tool [10, 11]. *Jikes* is a Java compiler system [6] and *Keystone* is a parser and front-end for ISO C++ [8, 14].

The rows of Table 1 list some statistics and coarse-grained size metrics for the test cases: the first row lists the version number, Version; the second row lists the number of source files, Source Files, for each test case; the third row lists the number of translation units, Translation Units, which includes both C++ and C translation units; the fourth row lists the number of C++ translation units (C++ Translation Units), which is only C++ code; and finally, the last row of the table lists the (approximate) thousands of lines of code (KLOC) for each test case, not counting blank or comment lines. For example, the largest game in our test suite is Freespace 2, a Version that we obtained from a cvs repository (on July 22, 2005), consisting of 652 source files, 220 Translation Units and 220

	Classes				Functions			
	Count	Abstract	Root	Leaf	Count	Member	Virtual	Pure
ASC	1389	58	901	390	8693	7775	2170	208
AvP	1732	28	1369	327	11548	9350	1216	90
Freespace2	332	0	320	12	9468	1687	48	0
Scorched3D	799	50	405	364	8432	7210	1907	112
Doxygen	315	9	153	157	5422	4570	2159	249
g^4re	78	17	27	37	849	798	303	106
Jikes	378	5	210	158	5717	5685	602	16
Keystone	160	14	49	87	2354	2306	1178	189

Table 2. *Modularity.* This table presents results about the modularity of game application software.

C++ Translation Units. Since the number of Translation Units is the same as the number of C++ Translation Units, the Freespace 2 test case contains no C code. The Freespace 2 test case consists of 365 *KLOC*, as illustrated on the last row, third column of Table 1.

The results in Table 1 suggest that, for the test cases that we have chosen for our study, the game applications are larger than the language processing applications. For example, the average number for the game applications is 231 KLOC, whereas the average number of KLOC for the language processing applications is 78 KLOC; thus, the average game application in our test suite is three times as large as the average language processing application.

4.2. Modularity in Game Application Software

Table 2 presents results that capture information about the modularity of game application software. The rows in Table 2 list the test cases, where the game applications are listed in the first four rows and the language processing tools are listed in the last four rows. The columns summarize results describing the classes and functions in the respective test cases: the first four columns summarize information about the classes and the last four columns summarize information about the functions. For example, the Alien vs Predator (AvP) game, listed on the second row of the table, contains 1732 classes: 28 of the classes are abstract, Abstract, 1369 of the classes contain no ancestors in the inheritance tree, Root, and 327 of the classes contain no children and have at least one ancestor in the inheritance tree, Leaf. The number of Root and Leaf classes for the AvP game together comprise 1696 classes so that the inheritance tree contained only 36 interior classes. Moreover, the (AvP) game contains 11,548 functions, Count, 9350 of these functions were members of a class, Member, 2170 of these functions were virtual, Virtual, and 208 of these functions were pure virtual, Pure.

The AvP game contains more classes than any other application in our test suite; the second highest class total is the ASC game containing 1389 classes. It is somewhat surprising that the AvP game contains a large number of classes in view of the large number of C files in the program. The results listed in Table 1 show that the AvP game contains 222 Translation Units, but only 95 of these are C++ Translation Units; which means that over half of the translation units consist of C code. However, the g^4re tool uses an Abstract Syntax Graph (ASG) representation of the input application; at the level of the ASG, all template classes are instantiated so that the count of classes listed in the first column of data in Table 2 includes classes and instantiated template classes and all of these structures may participate in inheritance and appear anywhere in an inheritance tree constructed for the Depth of Inheritance Tree, Number of Children and Number of Ancestor metrics.

The results in Table 2 suggest that, for our test suite of four game applications and four language processing tools, the game applications may be more modular than the language processing tools. For example, the total number of classes in the four game applications is 4252 classes. The results in Section 4.1 show that the total KLOC for the four game applications is 923 KLOC, for a classes to KLOC ratio of 4.6. Similarly, the total number of classes in the four language processing tools is 931 classes. The results in Section 4.1 show that the total KLOC for the four language processing tools is 310 KLOC, for a classes to KLOC ratio of 3.0.

4.3. Inheritance in Game Application Software

Tables 3 and 4 present results for the *Depth of Inheritance Tree* (DIT) and *Number of Children* (NOC) metrics respectively. Due to space limitations, we have elided the results for the *Number of Ancestors* (NOA) metric; the interested reader may consult reference [7] for results on the NOA metric. The rows in Tables 3 and 4 list the test cases. The columns list results for the metrics, where the

	Min	Max	Mean	Std Dev	Median	Mode
ASC	0	4	0.6451	0.9906	0	0
AvP	0	5	0.3043	0.7068	0	0
Freespace2	0	1	0.0361	0.1869	0	0
Scorched3D	0	9	0.9312	1.4957	0	0
Doxygen	0	4	0.9270	1.1220	1	0
g^4re	0	5	1.8974	1.5921	2	0
Jikes	0	3	0.7143	0.9596	0	0
Keystone	0	5	1.7938	1.4498	2	0

Table 3. *Depth Of Inheritance Tree.*

	Min	Max	Mean	Std Dev	Median	Mode
ASC	0	27	0.3139	1.5496	0	0
AvP	0	30	0.1207	0.8497	0	0
Freespace2	0	12	0.0361	0.6586	0	0
Scorched3D	0	11	0.2003	0.9122	0	0
Doxygen	0	48	0.3587	2.7672	0	0
g^4re	0	6	0.4872	1.1705	0	0
Jikes	0	26	0.4550	2.4741	0	0
Keystone	0	10	0.6063	1.6563	0	0

Table 4. *Number of Children.*

	Min	Max	Mean	Std Dev	Median	Mode
ASC	0	561	12.9770	30.3646	4	0
AvP	0	107	7.2898	10.5944	3	3
Freespace2	0	123	6.6596	15.7072	3	3
Scorched3D	0	240	17.3717	19.0581	12	3
Doxygen	0	430	27.6762	57.4967	7	7
g^4re	0	206	17.7564	30.2694	13	0
Jikes	0	2016	32.3968	119.1240	13	10
Keystone	0	557	24.3875	52.4735	15	14

Table 5. *Weighted Methods per Class.*

first three columns list the minimum, Min, the maximum, Max and the mean, Mean, values for the respective metrics. The final three columns in the tables list the standard deviation from the mean, Std Dev, the median, Median and the mode, Mode. The median value is the value for which an equal number of values lie above and below the median; the mode is the most common value.

The DIT metric measures the length of the maximum path from a class to the root of the inheritance tree and the NOC metric measures the number of immediate successors of a class in the inheritance tree. Intuitively, the DIT metric measures the depth of the inheritance tree and the NOC metric measures the breadth of the inheritance tree.

4.3.1. Depth of Inheritance (DIT). Table 3 presents the results for DIT that we computed for the eight test cases. For example, the fourth row of Table 3 lists DIT results for the Scorched3D game with a minimum depth of 0 and a maximum depth of 9, the longest inheritance depth of all of the test programs. Moreover, the Scorched3D game had the largest mean value, 0.09312, and the largest standard

deviation, 1.4957, so that this test case had the largest variance in the depth of its inheritance tree. The median and mode for the Scorched3D game are listed in the last two columns of the fourth row and these values are zero, as they are for all of the game application test cases.

4.3.2. Number of Children (NOC). Table 4 presents the results for NOC that we computed for the eight test cases. For example, the second row of Table 4 lists NOC results for the Alien vs Predator game (AvP) with a minimum number of children of 0 and a maximum number of children of 30, the broadest inheritance tree of all of the test programs. The AvP game had a mean value of 0.1207, and a standard deviation of 0.8497, the largest variance in NOC for any of the test cases. The median and mode for the AvP game are listed in the last two columns of the second row and these values are zero, as they are for all of the game application test cases.

The results in Table 4 suggest that the inheritance trees for language processing tools are broader than the inheritance trees for game applications. The average of the Max values for language processing tools is 22.5 classes and the average of the Max values for game applications is 20 classes. However, the classes to maximum breadth ratio of the language processing tools is 41.37 and for the language tools is 212.6.

4.4. Complexity in Game Application Software

Table 5 presents results for the *Weighted Methods per Class* (WMC) metric. The rows in the table list the test cases. The columns list results for the WMC metric, where the first three columns list the minimum, Min, the maximum, Max and the mean, Mean, values for weighted methods. The final three columns in the tables list the standard deviation from the mean, Std Dev, the median, Median and the mode, Mode.

	Allegiance	Civ:CtP2	BoB	Freespace2
Source Files	1135	2081	1132	649
Classes	1938	1963	715	110
DIT	9	7	11	7
NOC	49	312	123	39
LOC($\approx$)	331K	665K	451K	416K

Table 6. *Testsuite of DirectX Game Application Software.*

The results in Table 5 show that the methods in the language processing tools are more complex than the methods in the game application software. For example, the average maximum value of the language processing tools is 802.25, whereas the maximum value of the game applications is only 257.75. Similarly, the average Mean value for the language processing tools is 25.55, whereas the avereage Mean value for the game applications is only 11.07.

4.5. Results for the DirectX API

In this section we describe some preliminary results that we obtained for four commercial games implemented with the DirectX API. Our ongoing work includes the construction of an analysis system for the Windows platform that will enable us to automate our analysis and metric computation. The results in this section were computed by hand and we present them to provide some basis for comparison with the SDL game application software and the language processing tools presented in the first part of this section.

Table 6 lists four games implemented with the DirectX API. The games are *Allegiance* (ALLEGIANCE) from Microsoft Research, *Civilization: Call to Power 2* (CIV:CTP2) from Activision, *Battle of Britain* (BOB) from Empire Interactive and *Descent: Freespace 2* (FREESPACE2) from Volition Inc. The first row of the table contains information about the number of source files and the second row lists the number of classes. In computing the number of classes we counted template classes and not their instantiations. The third row of Table 6 lists the maximum depth of the inheritance tree (DIT) and the fourth row lists the maximum number of children (NOC). Finally, the last row of the table lists the lines of code KLOC, not including blank or comment lines.

5. Related Work

The literature on object-oriented software metrics is extensive; see, for example, the detailed survey presented in [3] and [5]. By contrast, there has been relatively little work focused on applying metrics to assess the relative design characteristics of systems in different application domains. Further, our work is unique in its consideration of gaming applications. Other researchers have, however, considered comparisons that are similar in spirit to our own. In this section, we briefly consider several relevant studies.

Paulson *et al.* [17] perform an empirical evaluation of the differences between open-source and proprietary software. Their goal is to evaluate the validity of common perceptions regarding open-source projects. Their study considers five dimensions of comparison: (*i*) system growth, (*ii*) design creativity, (*iii*) complexity, (*iv*) reliability, and (*v*) modularity. For each dimension of comparison, they apply a series of metrics to a testsuite consisting of three open-source projects and three proprietary projects. Based on the resulting figures, the authors conclude that relative to their proprietary counterparts, open-source projects (*i*) do not grow faster, (*ii*) foster more creativity, (*iii*) are more complex, (*iv*) are more reliable, and (*v*) are less modular. We note that we share the authors' interest in complexity and modularity, and use a similar metric for evaluating complexity.

MacCormack *et al.* [13] focus on the modularity of open-source software. Their approach is novel in its use of metrics defined over *design structure matrixes* [24]. Each matrix captures the dependencies between the source files in a given implementation. A value of one at position (i, j) denotes the existence of a call from a function defined in file i to a function defined in file j. Similarly, a zero value denotes the absence of such a call. The authors consider two metrics. The first metric estimates the number of files affected, on average, by an arbitrary system change. The second metric is based on a clustering algorithm that groups collections of interdependent files. The metric estimates the cost of coordination between the individuals responsible for implementing the various elements by allocating a higher cost to *inter*-cluster dependencies, and a lower cost to *intra*-cluster dependencies. When applied to their testsuite, which consists of one open-source system and one proprietary system, the resulting figures contradict the results of Paulson *et al.*: the open-source system appears more modular. Later, however, the authors evaluate a major redesign of the proprietary system which fares better: it is more modular than the open-source system. We note that the size of the testsuite makes it difficult to draw definitive conclusions.

6. Conclusions and Future Work

In this paper we have evaluated the exploitation of object technology in a test suite of game application software. We used metrics that describe the size, modularity, delegation, inheritance, and complexity of the applications. We developed a metric computation system that uses the g^4re infrastructure to analyze the C++ programs [10, 11], and we use the system to apply the metrics to the game application software and a test suite of language processing tools to form a basis of comparing the applications in the two domains. Our game applications are implemented using the SDL [18] and DirectX APIs.

We have shown that, for our test suite, the game applications are more modular than the language processing tools and that the methods in the game applications are less complex than the methods in the language processing tools. Our metrics computation system does not include library code in its analysis and it is likely that much of the complexity of the methods is game applications lies in the SDL or DirectX APIs. Our future work includes modification of the g^4re infrastructure to extrapolate the number of template classes rather than the number of instantiations of template classes. We are also extending the g^4re infrastructure to automate computation of the DirectX metrics.

References

[1] American National Standards Institute. *International Standard: Programming Languages - C++*. Number 14882:1998(E) in ASC X3. ISO/IEC JTC 1, September 1998.

[2] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. In *The second annual object-oriented numerics conference (OON-SKI)*, pages 122–136, Sunriver, Oregon, USA, 1994.

[3] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.

[4] S. F. Cohen. Quest for Java. *Communications of the ACM*, 41(1):81–83, January 1997.

[5] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 1998.

[6] IBM Jikes Project. Jikes version 1.22. Available at http://jikes.sourceforge.net.

[7] A. C. Jamieson, N. A. Kraft, J. O. Hallstrom, and B. A. Malloy. A metric evaluation of game software. Technical report, Clemson University, 2005.

[8] Keystone Project. Keystone version 0.2.3. Available at http://keystone.sourceforge.net.

[9] Gregory Knapen, Bruno Lague, Michel Dagenais, and Ettore Merlo. Parsing C++ despite missing declarations. In *7th International Workshop on Program Comprehension*, Pittsburgh, PA, USA, May 5-7 1999.

[10] N. A. Kraft, B. A. Malloy, and J. F. Power. g^4re: Harnessing gcc to reverse engineer C++ applications. In *Seminar No. 05161: Transformation Techniques in Software Engineering*, Schloss Dagstuhl, Germany, April 17-22 2005.

[11] N. A. Kraft, B. A. Malloy, and J. F. Power. Toward an infrastructure to support interoperability in reverse engineering. In *Proceedings of the 12th Working Conference on Reverse Engineering, WCRE'005*, Pittsbrugh, PA, November 2005.

[12] John Lilley. PCCTS-based LL(1) C++ parser: Design and theory of operation. Version 1.5, February 1997.

[13] A. MacCormack, J. Rusnak, and C. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. Working Paper 05-016, Harvard Business School, Boston, MA, USA, 2004.

[14] B. A. Malloy, T. H. Gibbs, and J. F. Power. Decorating tokens to facilitate recognition of ambiguous language constructs. *Software, Practice & Experience*, 33(1):19–39, 2003.

[15] I. Parberry. *Learn Computer Game Programming with DirectX 7.0*. Woodward Publishing Co., Plano, TX, USA, 2000.

[16] Ian Parberry, Timothy Roden, and Max B. Kazemzadeh. Experience with an industry-driven capstone course on game programming: extended abstract. *SIGCSE Bull.*, 37(1):91–95, 2005.

[17] J.W. Paulson, G. Succi, and A. Eberlein. An empirical study of open-source and closed-source software products. *IEEE Transactions on Software Engineering*, 30(4):246–256, 2004.

[18] E. Pazera. *Focus on SDL*. Premier Press, Cincinnati, OH, 2003.

[19] J. F. Power and B. A. Malloy. Metric-based analysis of context-free grammars. In *Proceedings of the 8th International Workshop on Program Comprehension*, Limerick, Ireland, June 2000.

[20] J. F. Power and B. A. Malloy. Symbol table construction and name lookup in ISO C++. In *37th International Conference on Technology of Object-Oriented Languages and Systems, (TOOLS Pacific 2000)*, pages 57–68, Sydney, Australia, November 2000.

[21] S.P. Reiss and T. Davis. Experiences writing object-oriented compiler front ends. Technical report, Brown University, January 1995.

[22] J.A. Roskind. A YACC-able C++ 2.1 grammar, and the resulting ambiguities. Independent Consultant, Indialantic FL, 1989.

[23] S. Singhal and B. Nguyen. The Java factor. *Communications of the ACM*, 41(6):34–37, June 1998.

[24] D.V. Steward. The design structure system: A method for managing the design of complex systems. *IEEE Transactions on Engineering Management*, 28(3):71–84, 1981.

[25] P. Tyma. Why are we using Java. *Communications of the ACM*, 41(6):38–42, June 1998.

[26] D. van Heesch. Doxygen version 1.3.9.1. Available at http://stack.nl/ dimitri/doxygen.